Un-preprocessing: Extended CPP that works with your tools*

Yufeng Cheng[1,2], Meng Wang[3], Yingfei Xiong[1,2], Zhengkai Wu[1,2], Yiming Wu[1,2], Lu Zhang[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, School of EECS, Peking University, China
[3]School of Computing, University of Kent, UK
xiongyf,zhanglucs@pku.edu.cn,m.w.wang@kent.ac.uk

ABSTRACT

Many tools directly change programs, such as bug-fixing tools, program migration tools, etc. We call them *program-modification tools*. On the other hand, many programming languages use the C preprocessor, such as C, C++, and Objective-C. Because of the complexity of preprocessors, many program-modification tools either fail to produce sound results under the presence of preprocessor directives, or give up completely and deal only with preprocessed code.

In this paper we propose a lightweight approach that enables program-modification tools to work with the C preprocessor for free. The idea is that program-modification tools now simply target the preprocessed code, and our system, acting as a bidirectional C preprocessor, automatically propagates the changes on the preprocessed code back to the un-preprocessed code. The resulting source code is guaranteed to be correct and is kept similar to the original source as much as possible. We have evaluated our approach on Linux kernel with a set of generated changes. The evaluation results show the feasibility and effectiveness of our approach.

ACM Reference format:
Yufeng Cheng[1,2], Meng Wang[3], Yingfei Xiong[1,2], Zhengkai Wu[1,2], Yiming Wu[1,2], Lu Zhang[1,2]. 2017. Un-preprocessing: Extended CPP that works with your tools. In *Proceedings of Internetware'17, Shanghai, China, September 23, 2017,* 10 pages.
https://doi.org/10.1145/3131704.3131715

1 INTRODUCTION

Many programming languages are provided with preprocessors [7, 18, 22]. The most widely used of all is the C preprocessor (CPP), forming part of C, C++, and Objective-C. CPP is lexical, which operates on tokenized source prior to any parsing. As a result, it is not restricted to a particular syntax and can be used casually in many languages as a general-purpose tool. For example, as an HTML authoring tool [19], CPP may be used to capture shared code pieces as macros.

*This work is supported by National Key Research and Development Program. 2016YFB1000105, the National Natural Science Foundation of China under Grant No.61421091, 61332010 and 61672045. Yingfei Xiong is the corresponding author.

Internetware'17, September 23, 2017, Shanghai, China

ACM ISBN 978-1-4503-5313-7/17/09…$15.00
https://doi.org/10.1145/3131704.3131715

This popularity of CPP causes a difficulty in program modification tools, such as program-repair tools [20, 21, 31, 36] and API evolution tools [23, 30, 35], where direct modification of source code is involved. Such tools typically do not handle C preprocessor directives. This is not only in the case for casual uses, where the target languages bear no connection to C. We have investigated the implementations of three influential bug-fixing tools on the C programming language: GenProg [20, 21], RSRepair [36], and SemFix [31], and all of them work only on preprocessed code. Users have to manually inspect the preprocessed code, and copy the changes to the original code—risking of introducing bugs in the process.

In this case, we are confronted with a classic problem in compilation, in fact any program transformation, that the transformation is one directional. Changes to the source are kept and propagated to the target when the transformation is run, but there is no obvious way to map changes of the target back to the source. This means poor usability, as the tool outputs are obfuscated to programmers who work on the source. Moreover, any change to target locks the source: there is no point of fixing a bug in the preprocessed code and only to have it overwritten when the source is compiled again.

Despite the evident necessity, the mapping from target to source is not an easy task, as the tools must be able to understand both the preprocessor directives and the target programming languages, and make sure whatever changes made on both levels are consistent with each other. A closely related area is refactoring [12, 28], where tools are expected to directly manipulate preprocessor directives. For example, one may well want to rename a macro or extract a macro as part of the refactoring. In this case, tool builders have no choice but to bite the bullet and confront the preprocessor directly. Typically a new C grammar is designed such that it incorporates both the original C grammar and the preprocessor directives. However, when applied to a more wider range of code editing tools, such almost brute force approaches exhibit obvious shortcomings. First, tool developers using such a grammar basically have to start from scratch: they have to learn the new grammar and leave behind the existing tool chains on C. Second, the effort spend on the new grammars is specialized and cannot be reused for other languages, which basically rules out casual uses of CPP.

In this paper we propose a lightweight approach to support CPP in program-modification tools. Our system acts as a bidirectional CPP: the original preprocessing can be considered as a forward program transformation, and we add to it a corresponding backward transformation that maps changes on the preprocessed code back into the unpreprocessed source. As a result, tool builders can now focus on what they set out to do, and have the results automatically mapped to the source. We list a few examples here: 1) as mentioned above the implementations of the three state-of-the-art bug-fixing approaches only deal with preprocessed code; 2) the

API evolution tools mentioned previously can also be implemented more conveniently by only dealing with preprocessed code; 3) all program-modification tools on languages that do not formally rely on CPP naturally fall into this category because the programs may be put under casual uses of CPP.

To sum up, our paper makes the following contributions:

- We propose a lightweight approach to handling the C preprocessor in program-modification tools based on bidirectional transformations. We analyze different design alternatives and present our design decisions, as well as defining the correctness laws (Section 2).
- We propose the first bidirectional algorithm that supports the change of transformation steps. The current algorithm is specifically designed for CPP, but we believe the idea is general and can be applied to other systems whose backward transformation may break existing transformation steps (Section 3).
- We evaluate our approach on the Linux kernel and compare our approach with two baseline approaches: one reflecting changes by copying back the entire changed file and one reflecting changes by copying back the changed lines. We also compared our approach with the variant which does not break existing transformation steps. The evaluation results show that our approach breaks much less macro invocations, and always produces correct results while the other two do not (Section 4).

Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

It might be worth mentioning that a *non-goal* of this paper is to support software product lines written in CPP. In this work, we focus only on one product but not a family of products, because of the following reasons. (1) There is a wide range of applications of CPP outside software product lines, which we target (2) It is difficult, if not impossible, to know how to propagate the changes on one product to other products in general, making any approach domain-specific.

2 BACKGROUND AND OVERVIEW

We begin with an overview of the C preprocessor, and discuss the high-level intuition of our approach.

2.1 The C Preprocessor

Table 1 shows the main preprocessor directives and operators. A preprocessor directive starts with a # at the beginning of the line and ends at the end of the line. There are four main types of preprocessor directives: `#pragma` providing compilation options, `#include` for including header files, `#if` for conditional compilation, and `#define` for macro definitions. Additionally, within a macro definition, we can use operators such as `##` and `#`, for concatenating two variables or quoting a variable. Finally, there are some pre-defined macros such as `__FILE__`, which will be replaced by the current values of the fields.

When the C preprocessor processes a source file, it transforms the source file in the textual order according to the following rules:

- The `#include` and `#if` directives are first expanded, and then the expanded token sequences are scanned.
- For each macro invocation, the arguments are first preprocessed, and then the invocation is expanded.
- If an argument contains # or ##, the unpreprocessed argument is used, otherwise the preprocessed argument is used.
- After a macro invocation is expanded, the expanded token stream is scanned again, where any newly introduced macro invocations are again expanded.
- To avoid infinite expansion, if a macro has been expanded during the expansion process, it will not be expanded again.
- Any new preprocessor directives produced in the expansions will not be used in processing.

```
1 #if BIGENDIAN
2   #define BYTE4 0
3 #else
4   #define BYTE4 3
5 #endif
6 #define set_zero(number, byte, bit)
  *((char*)&(number)+byte) &= ~(0x1 << (bit))
7 float x = -100;
8 set_zero(x, BYTE4, 8);
```

Figure 1: An example of code preprocessing

As a concrete example, let us consider the code in Figure 1, which is taken from the standard C math library with some simplifications. This piece of code intends to compute the absolute value of `x`. It contains a conditional-compilation directive (with each branch an object-like macro definition), a function-like macro definition (note that line 6 is wrapped to fit the width), and a macro invocation in line 8. When the code is scanned by the preprocessor, first the arguments of `set_zero` are processed and `BYTE4` is expanded into `3` on an Intel machine. Then the macro invocation to `set_zero` is expanded, leading to the following code:

```
float x = -100;
*((char*)&(x)+3) &= ~(0x1 << (8))
```

2.2 Backward CPP

Now suppose that a program-modification tool detects that the shifting of 8 times in the preprocessed code is wrong and changed it to 7 as below:

```
float x = -100;
*((char*)&(x)+3) &= ~(0x1 << (7))
```

Most program-modification tools will simply stop here and leave the modified code as it is. But this means we loose the modularity and portability of the original code, a situation that is undesirable at best.

Our system is designed to map such code back to the unpreprocessed source without losing the modifications made. That is to say, when the new source is preprocessed again, we will get back exactly the modified code — a property known as *round-tripping*. In the above example, we will trace back the preprocessing steps and fold back the macro expansions in the inverted order to produce the following.

Table 1: Main preprocessor directives and operators

Directives	Functionality	Example	Result
`#pragma`	Compiler options	`#pragma once`	removed from the preprocessed file
`#include`	File Inclusion	`#include <stdio.h>`	the content of "stdio.h"
`#if, #ifdef, . . .`	Conditional compilation	`#ifdef FEATURE1 x=x+1; #endif`	`x=x+1;`
`#define X`	Object-like macro definition	`#define X 100 a = X;`	`a = 100;`
`#define X(a, b)`	Function-like macro definition	`#define F(x) x*100 F(10);`	`10*100;`
`a ## b`	Concatenation	`#define X a_##100 X`	`a_100`
`#b`	Stringification	`#define F(x) #x; F(hello);`	`"hello";`
`__FILE__, __DATE__, ...`	Predefined macros	`__FILE__`	main.c

```
#if BIGENDIAN
  #define BYTE4 0
#else
  #define BYTE4 3
#endif
#define set_zero(number, byte, bit)
*((char*)&(number)+byte) &= ~(0x1 << (bit))
float x = -100;
set_zero(x, BYTE4, 7);
```

It is easily verifiable that the correction to the preprocessed code is made 'durable' now. Any further preprocessing will simply produce exactly the same corrected code.

This backward process is not always as straightforward. Since we permit arbitrary changes to proprocessed code, it is well possible that some of them may get in the way. Let us consider a variant of the example in Figure 1.

```
#if BIGENDIAN
  #define BYTE4 0
#else
  #define BYTE4 3
#endif
#define set_zero(number, byte, bit)
((char)&(number)+byte) |= ~(0x1 << (bit))
float x = -100;
set_zero(x, BYTE4, 7);
```

This time the bug is in the macro itself where the bitwise AND is erroneously written as an OR.

In this case, there are two options: 1. change the definition of the `set_zero` macro, 2. leave the `set_zero` macro expanded, while folding other macro expansions. The former makes global changes, affecting all invocation of `set_zero`, whereas the latter makes only local changes, affecting the one expansion. Our philosophy is to be conservative. Since it is in general unclear whether a local change is meant to affect all similar sites, we do not change macro definitions.

Design Decision 1. *Backward transformation shall not change any macro definition.*

Accordingly, the macro `set_zero` will remain expanded in the new source and its definition unchanged. But this does not mean that we will give up on restoring modularity and portability. Instead, we continue to fold the expansion of `BYTE4` and restore the conditional-compilation directive as the following

```
#if BIGENDIAN
  #define BYTE4 0
#else
  #define BYTE4 3
#endif
#define set_one(number, byte, bit)
((char)&(number)+byte) |= 0x1 << (bit)
float x = -100;
((char)&(x)+BYTE4) &= ~(0x1 << (7))
```

which leads to our second design decision.

Design Decision 2. *Backward transformation should aim to keep existing macro invocations and not to leave them expanded.*

Some might be tempted to go a step further by abstracting the new code into a new macro so that an invocation (of a different macro) will remain in place. This is too radical in our opinion as the defining of new macros may have unforeseen global consequences. Moreover, there is no evidence that such an effort will actually improve the quality of the resulting code. We think creative undertakings, such as designing new abstractions, is best left to human programmers, and thus our third design decision.

Design Decision 3. *Backward transformation shall not introduce new macro invocations.*

In addition to the three design decisions, our system satisfies two correctness laws that establish the round-tripping property. We call unpreprocessed code *source* and preprocessed code *target*.

Law 1 (FwdBwd). *If there is no change to target, backward transformation will not change the source.*

Law 2 (BwdFwd). *Preprocessing source produced by backward transforming a target will give back the same target.*

We will formally define the above laws in Section 3.4, when details of our approach is discussed.

2.3 Naive Solutions

For comparison, we list two baseline solutions. Despite being obvious crude and unfit for purpose, they are more or less what are available to tool users at the moment.

Naive solution I (per-file). The first naive solution is to directly copy back the changed preprocessed files and replace the original unpreprocessed files. This solution is the easiest to implement but has two major deficiencies. First, the original unpreprocessed source files may contain macro definitions, which will be lost if we directly copy back the preprocessed files, causing problems elsewhere. Second, this will leads to the expansion of all macro invocations in the source file, as well as all **`#include`** directives, destroying modularity completely.

Naive solution II (per-line). The second solution is to use more fine-grained units, copying back only the changed lines. This is doable with the assumption that modern compilers keep a traceability relationship between the lines in the unpreprocessed files and the preprocessed files. For example, when GCC preprocess our running example, it will replace lines 1-7 into empty lines, and put the three statements expanded from line 9 in one line, so that a one-to-one correspondence between lines are kept.

Although the second solution has the benefit of not removing macro definitions, it still has problems. First, a lot of macros are still unnecessarily expanded. The situation becomes even worse when we consider tools that copy lines of code within source files, e.g., GenProg [21] copies statements between two positions in the source files to fix bugs. In such cases all macro invocations in the copied lines will be lost. Second, if the original macro invocations span several lines, the backward mapping produces wrong results. For example, suppose there is a line break within the original macro invocation, as follows.

```
RESIZE(GARRAY(2),
  100, FREE);
```

In GCC, the macro invocation will be expanded into two lines, where the first line contains the three expanded statements, and the second line is empty. As a result, in the backward transformation only the first line will be copied back, resulting in an incorrect program.

3 APPROACH DETAILS

In this section, we present our approach to bidirectionalize CPP in a more formal setting. For presentation clarity, we build the framework considering only a subset of CPP, namely macro definition and invocation, before extending the solution to full CPP.

Due to page limit, we shall consider a subset of C preprocessor: the only directive is **`#define`** and there is no # or ## operators in macro bodies. This model could be expanded to cover full C preprocessor.

3.1 Modelling forward preprocessing

We view program source as a sequence of tokens of type *Token* (a synonym for *String*), and there is a global environment (of type *Env*) containing a list of macro definitions that are in scope. Each token is additionally annotated with its own expansion history as a set of disabled macros to ensure termination. The augmented tokens have type *TokenS*.

We use a functional model for CPP. The preprocessing can be seen as a function from an environment and a token sequence to a token sequence. We represent a sequence as a list. An empty list is written as '[ ]' and ':' is used to 'cons' an element to a list. As a result, we can write `"C" : "C" : "P" : [ ]`, or simply `["C", "C", "P"]`. These lists are of type [*Token*] (note the overloaded use of [ ] both on the term and type level). Similarly, the list $[1, 2, 3]$ is of type [*Int*]. Appending two lists is through infix operation '+', so that `["C", "C"]` + `["P"]` gives `["C", "C", "P"]`.

$$
\begin{aligned}
&\mathit{forward} :: (\mathit{Env}, [\,\mathit{TokenS}\,]) \to [\,\mathit{TokenS}\,] \\
&\mathit{forward}\ (\_, [\,]) = [\,] \\
&\mathit{forward}\ (\mathit{env}, \mathit{ts}@(\mathit{skped} +\!\!+ \mathit{orgnl} +\!\!+ \mathit{rest})) = \\
&\quad \mathit{skped} +\!\!+ (\mathit{forward}\ (\mathit{env}', \mathit{prcssed} +\!\!+ \mathit{rest})) \\
&\qquad \textbf{where}\ (\mathit{env}', \mathit{prcssed}) = \mathit{step}\ (\mathit{env}, \mathit{ts})
\end{aligned}
$$

This declaration defines a function named *forward*, which has type $(\mathit{Env}, [\,\mathit{TokenS}\,]) \to [\,\mathit{TokenS}\,]$. There are two cases in the definition handling empty and non-empty token lists as inputs, which are disinguished by the patterns of the formal parameters.

Patterns are similar in syntax to expressions, but appear on the left-hand side of '=', which match the input and bind the variables within. For example, the patten [ ] matches the empty list, and the patten $(c : cs)$ matches a nonempty list with c bound to the head of the list and cs bound to the tail of the list. Base on the matching of the input to the patterns, the right clause is executed and the expression on the right hand of the = is returned. For the sake of brevity in presentation, we also make use of non-standard patterns involving the append operator +, which binds variables to different segments of an input list. For example, $\mathit{ts}@(\mathit{skped} +\!\!+ \mathit{orgnl} +\!\!+ \mathit{rest})$ says that the input list is divided into three parts namely *skped*, *orgnl* and *rest*, and collectively the whole list is named *ts*. With this pattern, we omit the operational details of how the list is divided. Lastly, the underscore _ in pattern is known as *wild-card*, which independently matches value, but each binds nothing.

The above definition of *forward* states (in the first clause) that if the token list is empty ([ ]), the the output is an empty list, regardless of the environment input. If the token list is non-empty (in the second clause), then through simple scanning and environment lookup, the list is divided into three parts, which are the part that doesn't require processing (*skped*), the first part requires processing (*orgnl*) and the rest of the tokens (*rest*). The algorithm then processes *orgnl*, through function *step*, into *prcssed* and a probably updated environment (the **where** clause introduces a local binding), before passing both to a recursive call.

This process is best explained through an example. Consider the following source,

```
#define inc(x) x++
inc(a)
```

which is processed into `a++`. We list the steps of *forward* execution below (The parts in {-_ -} are explanations.).

```
  forward ([ ], [ ] ++ ["#", "define"...]
                    ++ ["inc", "(", "a", ")"])
↪ {-Macro defined -}
  [ ] ++ forward ([inc], [ ] ++ ["inc", "(", "a", ")"]
                             ++ [ ])
↪ {-inc(a) is expanded into a++ -}
  [ ] ++ forward ([inc], ["a"] ++ [ ] ++ ["++"])
↪ {-a is skipped -}
  ["a"] ++ forward ([inc], ["++"] ++ [ ] ++ [ ])
↪ {++ is skipped. -}
  ["a", "++"] ++ [ ]
```

At each step, the current token sequence is splited into *skped*, *orgnl* and *rest*. Then *skped* is appended to the final output, and *orgnl* is processed to *prcssed*

Function *forward* models the behaviour of CPP, but it is not sufficient for building backward processing. *forward*'s return result does not contain all information necessary for reverting the process. Instead, we work with an enriched function which is similar to *forward* but keeps more processing information.

```
complement :: (Env, [TokenS]) → [Action]
complement (_, [ ]) = [ ]
complement (env, act.skped ++ act.orgnl ++ rest) =
  [act] ++ (complement (act.aftEnv, act.prcssed ++ rest))
    where act = step (env, tokens)
```

The function is called *complement* because it records information of the forward processing (all the actions), in addition to the target token sequence.

In the above example of `inc` macro, *complement* produces three actions, namely *defInc*, *ivkInc* and *skp*, corresponding to the three execution steps. Roughly speaking *defInc* introduces macro `inc` into the environment, *ivkInc* expands the macro invocation `inc(a)` and produces `a++`, and *skp* moves `a++` to the final output.

We treat the action type *Action* abstract and only specify it with a set of operations.

```
bfrEnv :: Action → Env
aftEnv :: Action → Env
skped :: Action → [TokenS]
orgnl :: Action → [TokenS]
prcssed :: Action → [TokenS]
dsbles :: Action → [Macro]
step :: (Env, [TokenS]) → Action
```

The *step* function is the construction function for actions, which performs a step of forward processing. We also use the short hand *act.xxx* (where *x* is an accesser function) instead of *xxx* (*act*). In general, an action may:

- (the cases of **`#define`** and **`#undef`**) update the environment from *bfrEnv* to *aftEnv*.
- (the cases of **`#if`** and macro invocation) process the token-sequence prefix into *prcessd*, which is then added to the front of the remaining token sequence for further processing. The original prefix is stored as *orgnl*. It may also extend the disabled macro list *dsbles* in the case of macro invocation to prevent recursive invocation.
- do nothing else but add the prefix the final output *skped*. (This happens when the prefix is 'ordinary' tokens, neither a directive nor a macro invocation)

A particular subtlety we hide behind the interface is the non-linear nature of function-like macro invocation: the arguments are processed first before substituted into the body, and therefore there is a number of sub-actions generated in supporting of the main action.

As a last point, function *forward* is subsumed by *complement* as it is simply the concatenation of all the skipped tokens of the actions:

```
forward (e, ts) = concat (complement (e, ts).skped)
```

3.2 Modeling changes

Program-modifications tools can change the program with a variety of operations. In this paper we consider a unified type of operation: replacement.

For each token in a sequence, it can be replaced by a sequence of tokens. For example, token `"a"` can be changed to `["b"]`. Note that since the replacing token sequence can be of arbitrary length (including zero), this notion of replacement naturally covers deletion. Furthmore, we can replace a token into itself plus additional tokens, thus insertion is covered as well.

We represent changes to a token sequence as a sequence of changes to individual token, with one-to-one matching of replaced token and the replacing token sequence. In this representation, if a token sequence *ts* is changed to a token sequence *ts′* with changes *cs*, we have *length* (*ts*) = *length* (*cs*) and *ts′* = *concat* (*cs*). The tokens in the replacing sequence also inherits the disable macro set from the replaced token.

3.3 Backward processing

Backward processing traverses, in the reversed order, the actions taken in the forward processing, and tries to map changes in the target to changes in the source.

```
backward :: ([Change], [Action], [Token]) → [Change]
backward (cs, [ ], _) = cs
backward (cs, a : racts, tgt) = backward (cs′, racts, tgt)
  where cs′ = fstSucc (res)
        res = [back (stgy, a, cs, tgt) | stgy ← strategies]
```

Function *backward* iteratively traverses the reversed action list (the second parameter), and succeeds when there is no more action left. The first parameter is the changed target but represented as changes, which is updated during execution. Another copy of the changed target is passed in as the third argument, which remains constant during execution and is used for correctness checking. For each action, the one-step-function *back* is called with different

strategies and the first succeeding one is returned as the result. The list comprehension $[\,back\ (stgy, a, cs, tgt) \mid stgy \leftarrow strategies\,]$ is conceptually similar to set comprehension, which applies *back* with each strategy and returns all the results as a list.

$$
\begin{array}{l}
back :: (Strategy, Action, [\,Change\,], [\,Token\,]) \rightarrow [\,Change\,] \\
back\ (stgy, act, pre +\!\!+ skped +\!\!+ prcssed +\!\!+ rest, tgt) = \\
\quad \textbf{if}\ check\ (act.bfrEnv, pre +\!\!+ new, tgt) \\
\quad \textbf{then}\ pre +\!\!+ new\ \textbf{else}\ fail \\
\qquad \textbf{where}\ new = skped +\!\!+ stgy\ (act, prcssed) +\!\!+ rest
\end{array}
$$

Function *back* finds the target segment that are the processing result of the action, and uses a given strategy, which is a function of type $(Action, [\,Change\,]) \rightarrow [\,Change\,]$, to construct a new source segment. Let us consider the inc example, which is reproduced below.

```
#define inc(x) x++
inc(a)
```

Forward processing produces three top level actions *defInc*, *ivkInc* and *skp*. Now consider the simple case of changing `a++` to `b++`. We have the following execution steps of `backward` (The part in {-_ -} are the splited sequence passed to *back*). We omit the third argument that is used for correctness checking for now.

```
  backward (["b","+","+"], [skp, ivkInc, defInc], _)
↪  {-[] ++ ["b","+","+"] ++ [] ++ [] -}
  backward (["b","+","+"], [ivkInc, defInc], _)
↪  {-[] ++ [] ++ ["b","+","+"] ++ [] -}
  backward (["inc","(","b",")"], [defInc], _)
↪  {-[] ++ [] ++ [] ++ ["inc","(","b",")"] -}
  backward (["#","define",...,
             "inc","(","b",")"], [], _)
↪
  ["#","define" ... "inc","(","b",")"]
```

The strategy passed to *back* decides how a new source segment can be constructed from the changed target segment according to the current action (i.e., How `b++` is turned into `inc(b)`).

The most obvious strategy of backward processing a macro expansion is to fold it. For object-like macros, this is simply to replace the body by the name of the macro. For function-like macros, we additionally need to extract the parts of the body that came from argument subsitution, recursively process those parts to recover the arguments, and finally replace the body with an invocation.

There is nothing non-standard of this folding process. But there are plenty of opportunities for it to get stuck. In fact, the simple strategy of folding back every macro expansion fails every time when a change affects the 'non-parameter' part of the macro body. For example, in the above if we have changed `a++` to `a--`, there is no way to fold it back to `inc`. Moreover, even if a macro folding is successful, we still need to check the validity of the result, which will be explained later.

To recover from the failure in macro folding, we can try to cancel the folding and leave the modified expansion as it is. For example, the following steps backward process `a--`.

```
  backward (["a","-","-"], [skp, ivkInc, defInc], _)
↪
  backward (["a","-","-"], [ivkInc, defInc], _)
↪  {-folding cancelled -}
  backward (["a","-","-"], [defInc], _)
↪  {-macro definition restored -}
  backward (["#","define",...,"a","-","-"], [], _)
↪
  ["#","define",...,"a","-","-"]
```

Note that instead of recreating a macro invocation, the expansion `a--` is left as expanded in the final result. A particular tricky part in the implementation of this folding cancellation is the maintaining of alignment between actions and their corresponding segments in the token list. But we do not go into details here.

This strategy of keeping macro invocations expanded may appear to be a silver bullet, avoiding failing of backward processing altogether. However, the fact that one may produce some source does not mean it is correct. For example, consider the following source.

```
#define L x L
L
```

Forward processing produces `x L`, and suppose it is changed to `y L`. Since the body of the macro is changed, there is no way to fold the expansion. However, leaving the expansion as it is is also wrong. If we forward process

```
#define L x L
y L
```

it will produce `y x L` instead of `y L`, violating the round-tripping law. This is because the disabled recursive invocation of `L` is accidentally enabled by the change.

There is no easy way to detect such problems by just looking at the changes made. We therefore employ a generate-and-check strategy: after every step of backward processing, we forward process the new source and verifies whether the same changed target is produced. This is where the third argument of *backward* is used.

$$
\begin{array}{l}
check :: (Env, [\,Change\,], [\,Token\,]) \rightarrow Bool \\
check\ (e, pre +\!\!+ new, tgt) = \\
\quad concat\ (pre) +\!\!+ (forward\ (e, concat\ (new))) \equiv tgt
\end{array}
$$

This checking mechanism guarantees correctness, but is not efficient: it takes at least linear time with respect to the number of actions. In the implementation, we perform certain optimisations to speed up the process. The details are omitted here for space reasons.

We can further improve this recovery strategy by making the cancellation of macro folding more delicate. For example, consider the following source program.

```
#define double(x) x+x
#define inc(x) ++x
double(inc(a))
```

Forward processing produces `++a + ++a`. If the change is `++a + ++b`, the folding of `double` will fail in the backward direction, and the simple strategy above will leave the expansion as it is.

```
#define double(x) x+x
#define inc(x) ++x
++a + ++b
```

But it is not difficult to see in this case that although the root action of expanding `double` has to be cancelled, its sub-actions handling the arguments can still be inverted. Therefore, our refined strategy returns

```
#define double(x) x+x
#define inc(x) ++x
inc(a)+inc(b)
```

Perhaps unsurprisingly, there are plenty of opportunities for things to go wrong here too. Let us consider this source program.

```
#define a(x) c
#define p (1)
#define inc(x) ++x
inc(a p)
```

In the forward process, `a p` will firstly be processed as `inc`'s argument with `p` expanded to `(1)`. Next, `inc` is expanded, and `a (1)` is substituted into `inc`'s macro body, resulting in `++a (1)`. Finally, `++a (1)` is preprocessed to `++c` with the expansion of macro `a`. In the backward direction for a modified target `+c`, we first fold the expansion of `a`. Then if we cancel the folding of `inc`, without cancelling that of `p`, we will get a new source `+a p`. This is wrong because forward processing the new source produces `+a (1)` instead of `+c`, as the `+a` part, not being the result of an expansion, will not be processed again despite the later replacement of `p` with `(1)`.

This is another example showing the intricacy in guaranteeing correctness, and the necessity of the check performed at each backward step. Our algorithm will realise the folding of `p` in the above described backward process is wrong, and will fall back to the default strategy of keeping the expansion, and return the correct result `+a (1)`.

3.4 Correctness

We are now in a position to formally define the rounde-trip laws introduced in Section . To convert between token lists to change lists, we use a function *idChg* that creates identity changes for its input where each token is changed to itself.

Property 1 (FwdBwd). *For any token list ts, let*
$acts = complement\ ([\,], ts)$
$tgt = forward\ ([\,], ts)$
Then
$backward\ (idChg\ (tgt), acts, tgt) \equiv idChg\ (ts)$.

□

Property 2 (BwdFwd). *For any token list ts and change list chg. Let*
$acts = complement\ ([\,], ts)$
$tgt = forward\ ([\,], ts)$
$chgS = backward\ (chg, acts, tgt)$
If chgS is not failure, then
$forward\ ([\,], concat\ (chgS)) \equiv tgt$.

□

Giving the correctness checks that are performed at each step of backward processing, we can see that the above properties hold.

4 EVALUATION

4.1 Research Questions

In this section we focus on the following research questions.

- **RQ1: Macro Preservation.** According to Design Decision 2, our approach aims to preserve existing macro invocations. How does the strategy perform on actual programs? How does it compare to other techniques?
- **RQ2: Correctness.** Our approach is guaranteed to be correct according to Law 1 and 2. How important is this correctness? How does our approach compare to other techniques that do not ensure correctness?
- **RQ3: Failures.** Our approach may report a failure when it cannot find a proper way to propagate the change. How often does this happen? Are the failures false alarms (there exists a suitable change but our approach cannot find it)?

To answer these questions, we conducted a controlled experiment to compare our approach with the two baseline approaches described in Section 2.3 on a set of generated changes on Linux kernel source code. Moreover, since a large part of effort is spend on the macro-folding cancellation strategies, we also compare our solution with the variant without cancellation to see whether the effort has paid off. In the rest of the section we describe the details of the experiment.

4.2 Setup

4.2.1 Implementation. We have implemented our approach in Java by modifying JCPP[1], an open source C Preprocessor. We also implemented the two naive approaches in Section 2.3 and the method without cancellation for comparison. Our implementation and experimental data can be found on our repository[2].

4.2.2 Benchmark. Our experiment was conducted on the Linux kernel version 3.19. We chose Linux source code because Linux kernel is one of the most widely used software projects implemented in C. It contains contributions from many developers, and has a lot of preprocessor directives and macro invocations.

To conduct our experiment, we need a set of changes on the Linux kernel code. Since we concern about how different backward transformations affect preprocessing, we generated changes only in functions that contain macro invocations. To do this, we first randomly selected 180 macros definitions from the kernel code. Since there are far more object-like macros than function-like macros, we would select very few function-like macros if we use pure random selection. So we controlled the ratio between object-like and function-like macros to be 1.5 : 1. Based on the selected macros, we randomly selected a set of functions which contain invocations to the macros. Finally, we randomly selected 8000 lines from the

[1] http://www.anarres.org/projects/jcpp/
[2] |https://github.com/harouwu/BXCPP|

functions. There are in total 133 macro invocations in the selected lines.

Next we generated a set of changes on the selected lines. To simulate real world changes, we randomly generated two types of changes. The first type is token-level change, in which we randomly replace/delete/insert a token. The second type is statement-level change, in which we delete a statement or copy another statement to the current location. These two types of changes are summarized from popular bug-fixing approaches [17, 21, 36]. The statement-level changes are directly used by GenProg [21] and RSRepair[36]. The token-level changes simulate small changes such as replacing the argument of a method or change an operators used in approaches such as PAR [17].

More concretely, we had a probability p to perform an operation on each token, where the operation is one of insertion, replacement and deletion, which had equal probability. The replacement was performed by randomly mutating some characters in the token. The insertion was performed by randomly copying a token from somewhere else. Similarly, we had a probability q to perform an operation on each statement, where the operation is copy or deletion. The copied statement was directly obtained from the previous statement. We recognized a statement by semicolon.

Different tools may have different editing patterns: a migration tool typically changes many places in a program, whereas a bug-fixing tool may change a few places to fix a bug. To simulate these two different densities of changes, we used two different set of probabilities. For the high-density changes, we set $p = 0.33$ and $q = 0.1$. For the low-density changes, we set $p = 0.1$ and $q = 0.05$.

We generated ten sets of changes, five with high-density and five with low-density. The number of the changes generated for each set is shown in Table 2.

Table 2: Changes generated for the experiment

Low Density	Set	1	2	3	4	5
	Changes	952	885	956	967	884
High Density	Set	6	7	8	9	10
	Changes	3133	3136	3088	3123	3048

4.2.3 Independent variables. We considered the following independent variables. (1) *Techniques*, we compared our approach with the two naive solutions, per-file and per-line and the variant without cancellation called no-cancellation. (2) *Density of changes*, we evaluated both on the five high-density change sets and the five low-density change sets.

4.2.4 Dependent variables. We considered two dependent variables. (1) *Number of remaining macro invocations*. We re-ran the preprocessor after the backward transformation, and counted how macro invocations are expanded during preprocessing. Since none of the techniques will actively introduce new macro invocations, the number of expanded invocations is the number of remaining invocations. To avoid noise from included files, we count only the macro invocations in the current file. (2) *Number of errors*. We re-ran the preprocessor, and compared the new preprocessed program with the previously changed program by Unix file-comparing tool fc. Every time fc reported a difference, we counted it as an error. (3) *Failures*. Our approach may fail to propagate the changes, and we record whether a failure is reported for each change set.

4.3 Threats to Validity

A threat to external validity is whether the results on generated changes can be generalized to real world changes. To alleviate this threat, we used different types of changes and different density of changes, in the hope of covering a good variety of real-world changes.

A threat to internal validity is that our implementation of the three approaches may be wrong. To alleviate this threat, we investigated all errors we found in the experiments, to make sure it is a true defect of the respective approach but not a defect in our implementation.

4.4 Results

Table 3: Experimental Results

Low Density	Set	1	2	3	4	5
Our Approach	Macros	73	75	72	80	81
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	23	25	23	20	26
	Errors	6	7	6	7	7
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0
No-Cancellation	Macros	71	72	70	78	79
	Failures	62	61	63	55	54
High Density	Set	6	7	8	9	10
Our Approach	Macros	47	51	53	48	44
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	9	7	7	8	10
	Errors	6	6	7	6	6
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0
No-Cancellation	Macros	46	49	51	47	43
	Failures	87	84	82	86	90

Row "Macros" shows the number of remaining macros. Row "Errors" shows the number of errors caused. Row "Failures" indicates whether a failure is reported in the backward transformation.

The result of our evaluation is shown in Table 3. We discuss the results with respect to the research questions below.

4.4.1 RQ1 Macro Preservation. As we can see, our approach preserves macro invocations. Per-line preserves very few macro invocations, while per-file, as we expected, preserves no macro invocations. It is interesting that macro invocations which no-cancellation preserves are just a bit less than our approach.

We further investigated why per-line preserves so few macro invocations. One main reason we found is that some other tokens usually come with the macro invocations on the same line and per-line will expand the macros when any tokens in this line is changed even if there is no token changed in the expansion of the macro.

Our system uniformly outperforms, in term of macro preservation, the simpler approach No-Cancellation. The difference happens when there is a nested invocation of macros (macro invocations inside the parameters of a macro invocation), and a change has lead to the folding of the main invocation to be cancelled. Then No-Cancellation fails, while our system succeeds with cancelling the main invocation, but preserves the sub-invocations. Such a combination of situations is not very common, but still notably present in all of our experiment sets.

4.4.2 RQ2 Correctness. Our approach, per-file and no-cancellation lead to no errors while several errors are caused by per-line. This is because there are a few macro invocations that cross multiple lines. These macros take expressions or statements as argument, which are usually too long to be included in one line. If a change is generated on any one of the lines, per-file produces an error. On the other hand, such macro invocations usually have a larger expanded form, and are more likely to be changed in our experiment. As a matter of fact, most of such invocations were changed in all change sets.

4.4.3 RQ3 Failures. As we can notice, no-cancellation reports a large number of failures, while the one with cancellation has none. This stark contrast shows that our carefully crafted cancellation, which is arguably a major contribution of our work, has achieved its intended purpose.

Though not shown in the experiment, it is known that our approach with cancellation may fail to backward process a target too. This is usually because the changes accidentally introduce a new macro invocation in the preprocessed code, where there is no way to satisfy the roundtrip laws. However, we do not observe any such cases in our experiment. The reason is that macros usually have special names and it is not easy to collide with a macro name. Note the two baseline approaches never report a failure, so their numbers are not included in Table 3.

Also note that theoretically our approach may report false alarms: our approach reports a failure but a correct change on the source program exists. For example, let consider the following code piece,

```
#define p (x)
plus p
```

where `plus` is a function-like macro. After preprocessing, this code piece becomes `plus (x)`. If we change the last parenthesis into `) hello`, our approach reports a failure because first `p` will be expanded and then the expanded content forms a new macro invocation with `plus`. However, there exists a feasible change: replacing `p` with `p hello`. Nevertheless, such cases are probably rare, and should not be a problem in practice.

5 RELATED WORK

5.1 Bidirectional Transformation

Our work is inspired by research on bidirectional transformation. A classical scenario is the *view-update problem* [2, 5, 6, 8, 14] from database design: a view represents a database computed for a source by a query, and the problem comes when translating an update of the view back to a corresponding update on the source.

Languages have been designed to streamline the development of such applications involving transformations running bidirectionally. Notably the *lenses* framework [9], covering a number of languages that provide bidirectional combinators as language constructs.

A different approach is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as *bidirectionalization* [27, 42]. In the software model transformation literature, the underlying data to be transformed are usually in the form of graphs (instead of trees), and a relational (as oppose to functional) approach that specifies the bidirectional mappings between different model formats is more common [15, 32, 37, 40] However, the requirement of our work goes beyond what these languages offer: in our framework, not only data, but also transformations (macros) are subject to bidirectional updates.

5.2 Analyzing and editing unpreprocessed C code

The C preprocessor poses a great challenge for static program analyses. The ability of producing a number of possible preprocessed variants causes a combinatorial explosion, rendering it infeasible to employee traditional tools that are designed to analyze a single variant at a time. Only until very recently, sound parsing and analyzing unpreprocessed C code is made possible through *family-based analyses* [13, 16, 25]. Earlier tools have to resort to unsound heuristics or restrict to specific usage patterns [3, 11, 34].

Similarly, a lot of efforts in refactoring C code are devoted into dealing with multiple variants. Most approaches [10, 12, 39, 41] try to find a suitable model that represent both the C program and the preprocessor directives. A recent approach [33] suggests an alternative: perform refactoring on one variant and prevent the refactoring if problems may be caused in other variants. This is based on the observation that changes on one variant seldom causes problems in other variant.

Unlike these approaches, our approach currently considers only one variant. In the future we may combine our approach with these approaches to deal with multiple variants. However, handling only one variant is already useful in many cases: (1) many programs, though with conditional compilation, do not have many variants; (2) as revealed by Overbey et al. [33], changes in one variant often do not cause problems in other variants.

5.3 Empirical studies on the C preprocessors

Over the years, there has been no shortage of academic empirical studies that are critical towards the C preprocessor [7, 24, 38], and replacements of CPP are proposed such as syntactical preprocessors [28, 43] and aspect-oriented programming [1, 4, 26] are plenty. However until present, there is no sign of any adoption of these alternatives in industry, with the C preprocessor is still being seen as the tool of the trade [29].

6 CONCLUSION

Handling the C preprocessor in program-modification tools is difficult, as a result many tools either produce unsound results or give up on handling CPP entirely. In this paper we show that we can separate the concerns by using bidirectional transformations to

deal with the preprocessor, so that program-modification tools may focus only on the preprocessed code, achieving a more modular design.

We have conducted an experiment with Linux kernel source to test the effectiveness of our approach. The result shows our system (1) maintains the modularity of the source by not expanding macro invocations unnecessarily, (2) always produces correct results, (3) and never fails.

REFERENCES

[1] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 243–254, New York, NY, USA, 2009. ACM.

[2] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 281–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 333–336, New York, NY, USA, 2010. ACM.

[5] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[6] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[7] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *Software Engineering, IEEE Transactions on*, 28(12):1146–1170, 2002.

[8] L. Fegaras. Propagating updates through xml views using lineage tracing. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 309–320, March 2010.

[9] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[10] Alejandra Garrido and Ralph Johnson. Challenges of refactoring c programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pages 6–14, New York, NY, USA, 2002. ACM.

[11] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a c program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 379–388, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Alejandra Garrido and Ralph Johnson. Embracing the c preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12):1285–1304, 2013.

[13] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of c by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 323–334, New York, NY, USA, 2012. ACM.

[14] Stephen J. Hegner. Foundations of canonical update support for closed database views. In Serge Abiteboul and Paris C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 1990.

[15] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 205–216. ACM, 2010.

[16] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

[18] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LISP and functional programming*, pages 151–161, 1986.

[19] Jukka Korpela. Using a c preprocessor as an html authoring tool. http://www.cs.tut.fi/~jkorpela/html/cpre.html, accessed on Jan 8, 2015, 2000.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13. IEEE, 2012.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[22] Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S McKinley. Marco: Safe, expressive macros for any language. In *ECOOP*, pages 589–613. Springer, 2012.

[23] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe java program adaptation between APIs. In *PEPM*, pages 91–102, 2015.

[24] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM.

[25] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.

[26] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 191–204, New York, NY, USA, 2006. ACM.

[27] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 47–58. ACM, 2007.

[28] Bill McCloskey and Eric Brewer. Astec: A new approach to refactoring c. In *ESEC/FSE-13*, pages 21–30, 2005.

[29] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, , and Rohit Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *ECOOP*, page to appear. 2015.

[30] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[31] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[32] Inc. (OMG) Object Management Group. Meta object facility (mof) 2.0 query/view/-transformation specification. https://hackage.haskell.org/package/lens.

[33] Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring c with macros. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 75–85, New York, NY, USA, 2014. ACM.

[34] Yoann Padioleau. Parsing c/c++ code without pre-processing. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *PLOS*, 2006.

[36] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265, 2014.

[37] Andy Schürr. Specification of graph translators with triple graph grammars. In ErnstW. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin Heidelberg, 1995.

[38] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with c news, 1992.

[39] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng.*, 29(11):1019–1030, November 2003.

[40] Perdita Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, 2010.

[41] M. Vittek. Refactoring browser with preprocessor. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 101–110, March 2003.

[42] Janis Voigtländer. Bidirectionalization for free! (pearl). In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009.

[43] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 156–165, New York, NY, USA, 1993. ACM.